Сравнение эффективности обходчиков UniTESK

А. Р. Арутюнян

Российско-Армянский (Славянский) государственный университет,
ул. Овсепа Эмина, 123, Ереван, Армения
E-mail: miollnyr@rambler.ru

Аннотация

В данной работе исследуется эффективность генерации тестов на основе автоматического построения обхода графа, т.е. маршрута, проходящего через все его дуги. Приводятся экспериментальные данные о работе на графах различных типов обходчиков UniTESK, использующих разные алгоритмы построения обхода — алгоритм поиска в глубину и жадный алгоритм. Жадный алгоритм в большинстве случаев строит более короткие маршруты, позволяя выполнять соответствующие тесты значительно быстрее.

1. Введение

В настоящее время тестирование на основе моделей получает все большее распространение. Используемые в его ходе модели могут быть более абстрактными, чем реализация, но в то же время они хорошо отражают основные особенности тестируемой системы. Относительно небольшой размер моделей позволяет реализовать их автоматическую обработку, а подобие модели тестируемой системе гарантирует, что тесты, сгенерированные на основе модели позволят провести ее систематическое тестирование. Кроме того, тесты, построенные на основе моделей, легче сопровождать и проще переиспользовать из-за их более высокого уровня абстракции и независимости от реализации.

Одним из наиболее успешных примеров применения подхода тестирования на основе моделей является технология UniTESK [1,2], разработанная в Институте Системного Программирования РАН. В рамках этой технологии математические модели используются для решения основных задач тестирования: оценки корректности поведения целевой системы, генерации тестовых данных, тестовых последовательностей и оценки качества тестирования.

Генерация тестовой последовательности в технологии UniTESK производится на основе построения обхода графа, выступающего в качестве абстрактной модели тестируемой системы и отражающего выделенные разработчиком тестов аспекты ее поведения. Уникальной особенностью технологии UniTESK является задание графа модели в *неизбыточном* виде. Это означает, что перед началом тестирования о графе ничего не известно и вся информация о нем появляется только в процессе

тестирования. Таким образом, граф может не только задаваться в виде статической модели тестируемой системы, но и строиться динамически, в результате наблюдений за поведением тестируемой системы во время тестирования.

Извлечение модели тестируемой системы в процессе выполнения теста позволяет добиться существенного улучшения качества тестирования, масштабируемости тестового набора и упрощения переиспользования тестовых сценариев. Для обеспечения этих преимуществ все используемые алгоритмы работы с графом должны учитывать тот факт, что информация о графе появляется только во время выполнения теста.

Основными понятиями технологии UniTESK являются понятия спецификации, медиатора, тестового сценария и обходчика.

Спецификация представляет собой описание формальной модели тестируемой функции.

Медиатор осуществляет взаимодействие спецификации (формальной модели) и реализации тестируемой функции.

Тестовый сценарий представляет собой неизбыточное описание графа, моделирующего тестовую систему, и состоит из набора сценарных функций и функции вычисления текущего состояния — вершины графа. Каждая *сценарная функция* представляет совокупность однотипных тестовых воздействий.

Обходчик, основываясь на тестовом сценарии, осуществляет перебор всех допустимых тестовых воздействий во всех достижимых состояниях задаваемого этим сценарием графа.

Целью данной работы исследование эффективности алгоритмов обходчиков, входящих в инструмент CTesK, результаты которого позволят дат рекомендации по построению тестовых сценариев в условиях ограниченных ресурсов.

2. Сфера применения тестирования на основе спецификаций

Тестирование на основе спецификаций максимально эффективно при тестировании групп функций с объемным, логически связным текстуальным описанием, а также функций с закрытым кодом.

Рассмотрим преимущества подхода тестирования на основе спецификаций, на примере технологии UniTESK.

- *Абстрагирование от архитектуры*.
  Независимость тестов от внутренней архитектуры реализации становится возможной, благодаря отделению кода спецификации от медиаторов.
  В результате спецификации обладают самостоятельной ценностью. При переходе от одной архитектуры реализации к другой может потребоваться

изменение медиаторов, логика же тестирования остается неизменной. Разработчику тестов для системы, работающей на специфической платформе достаточно иметь спецификации и интерфейс реализации: они однозначно определяют построение медиаторной части.

- *Конфигурационная система.*
  Благодаря разделению спецификаций и реализационно-зависимых компонентов теста, возникает возможность создания гибкой конфигурационной системы, способной адекватно отражать в спецификациях неописанные варианты поведения функций. В частности, это относится к тем случаям, когда поведение тестируемой функции сильно зависит от специфики конкретной реализации (implementation-defined).
- *Генерация отчета по покрытию проверенных требований работы функции.*
  Такие отчеты, помимо оценки качества тестирования, дают возможность выделить часто используемые части программы, для их возможной дальнейшей оптимизации.
- *Разделение сценариев и спецификаций.*
  Такое разделение позволяет использовать одно описание функциональности тестируемой системы при ее тестировании в различных условиях.
  Например, сценарии легко дорабатываются в следующих направлениях.
  - *Увеличение множества тестовых значений параметров функции.*
    Это приводит к более полному тестированию функции. Обычно для минимальной полноты тестов требуется по одному набору значений параметров на каждую ветвь функциональности, определенную в спецификации данной функции.
  - *Вовлечение в тестирование функции связанных с ней функций.*
    Этот метод также приводит к более полному тестированию реализации функции.
    Особенно он важен при тестировании функций со скрытым состоянием, на которое можно влиять исключительно опосредованно, путем вызова других функций.

3. Как эффективность обходчика влияет на тестирование

Используемый в тестах, построенных по технологии UniTESK, обходчик позволяет осуществить полный перебор возможных тестовых ситуаций путем вызова всевозможных тестовых воздействий во всех состояниях графа, заданного в неизбыточном виде. Таким образом могут быть получены последовательности тестовых воздействий, которые практически никогда не проверяются при ручном построении тестов.

Может показаться, что для разработки хорошего теста достаточно указать набор сценарных функций и определить функцию вычисления состояния, и после этого обходчик все сделает сам. В общем случае это неверно. Неправильно выбранный метод обхода, неправильные ограничения на количество состояний, неправильное определение состояния графа: все это может привести к невозможности получения результата в условиях ограниченности ресурсов.

Возникает следующая проблема: ошибка может проявиться в последовательности тестовых воздействий, состоящей из сотни переходов по дугам графа, но выбранный механизм обхода будет не в состоянии обеспечить достижения подобной ситуации, по причине того, что он неудачно использует полный перебор.

Пример. Если человек находится на улице у дома номер 1, и хочет дойти к дому номер 91, то он может пойти прямо и дойти до цели. Действуя иначе, он может поворачивать на любом перекрестке, стараясь не проходить по два раза по одному и тому же кварталу. По дороге он увидит много интересного, но к цели придет с колоссальным запозданием.

Из этого следует, что строить обход графа, заданного в неизбыточном виде, нужно аккуратно. При неосмотрительном использовании перебора всех возможных вариантов в сложных ситуациях могут возникнуть трудности.

Поэтому детали реализации алгоритма обходчика имеют большое значение для эффективности тестирования больших систем. Очевидно, что построить обход всех дуг графа можно различными путями, необязательно эквивалентными по эффективности использования памяти и времени. Приведем несколько примеров.

«Цепной» пример. Описываемая в этом примере ситуация возникает в тех случаях, когда состоянием системы является целое число, и доступны два вида тестовых воздействий: увеличивающее и уменьшающее значение состояния на единицу.

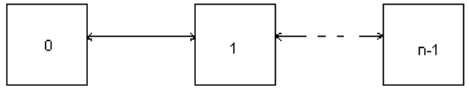

Рисунок 1. Граф состояний и переходов для «цепного» примера.

В этом случае возможен обход вида (перечисляются состояния в порядке их посещения) 0, 1, 0, 1, 2, 1, 0, …, 0, 1, 2, …, (n-2), (n-1), (n-2), …, 1, 0.

Также возможен обход вида 0, 1, 2, …, (n-2), (n-1), (n-2), …, 1, 0.

В обоих случаях проходятся все дуги во всех состояниях графа, однако в первом случае количество тестовых воздействий проделанных при обходе равно $2n(n+1)/2 = n(n+1)$, тогда как во втором оно составляет только $2n$.

Из приведенного примера, вытекает следующая рекомендация по работе обходчика: в случае наличия непройденных дуг из текущего состояния графа, обходчику рекомендуется первой вызывать операцию, соответствующую одной из этих непройденных дуг. То есть, не рекомендуется переходить к вызову сценарной функции B, пока не исчерпаны возможные вызовы сценарной функции A.

При этом порядок задания сценарных функций, определяющий последовательность их вызова обходчиком, должен тщательно обдумываться разработчиком тестов для определения максимально эффективного обхода.

Примеры с потоками. Приведенные ниже примеры выбраны из-за высокой степени связности графов, а также относительно малого количества состояний и дуг, что позволяет понять существо проблемы, не рассматривая сложные случаи.

О том, что такое *поток (thread)* и *обработчик завершения потока (cleanup handler)* можно прочесть в [3] и [4] соответственно.

Простой пример с потоками. Граф состояний и переходов простой модели системы управления потоками, описываемой в стандарте POSIX [5], изображен на Рисунке 2. В этой модели разрешается иметь не более одного потока одновременно.

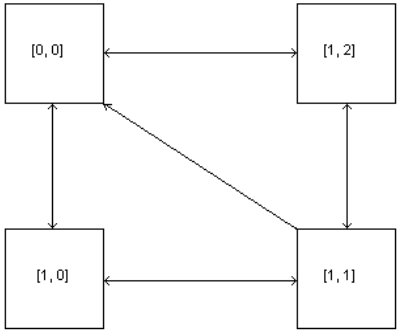

Рисунок 2. Граф состояний и переходов для простого примера работы с потоками.

Здесь первая цифра в состоянии обозначает количество активных потоков в системе, а вторая — количество зарегистрированных обработчиков завершения потоков.

В POSIX есть операции создания нового потока (обозначим ее как *C* от *c*reate), помещения функции в стек обработчиков завершения данного потока (*U*, от p*u*sh) и выталкивания ее из этого стека (*O*, p*o*p), а также операция уничтожения данного потока (*K*, *k*ill). В нашей простой модели потоку разрешается иметь не более двух обработчиков завершения.

В рассматриваемом случае, даже если обходчик следует рекомендации по использованию сценарных функций, существует, по крайней мере, два возможных маршрута обхода с разной эффективностью.

Первый вариант (13 переходов): [0, 0], [1, 0], [1, 1], [1, 2], [1, 1], [1, 0], [0, 0], [1, 0], [1, 1], [0, 0], [1, 0], [1, 1], [2, 1], [0, 0]. Он соответствует последовательности использования сценарных функций *CUOK*. Обходчик пытается в каждом состоянии вызывать функции в указанном порядке.

Второй вариант (15 переходов): [0, 0], [1, 0], [0, 0], [1, 0], [1, 1], [0, 0], [1, 0], [1, 1], [1, 0], [1, 1], [1, 2], [0, 0], [1, 0], [1, 1], [1, 2], [1, 1]. Он соответствует последовательности сценарных функций *CKUO*.

Сложный пример с потоками. Рассмотрим более сложную модель той же системы, допуская уже два активных потока.

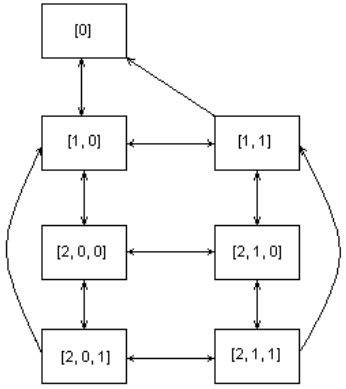

Рисунок 3. Граф состояний и переходов более сложной модели потоков.

В данном случае первая цифра в состоянии обозначает количество активных потоков. Далее идут несколько цифр, отражающих количество обработчиков завершения для соответствующих потоков.

В системе имеются те же операции, что и раньше, но на этот раз разрешим потоку иметь не более одного обработчика завершения.

В данном случае, даже если рекомендация по последовательному использованию сценариев выполняется, существует, по крайней мере, два возможных пути, с разной эффективностью.

Первый вариант (25 переходов, соответствует порядку *CKUO*): [0], [1, 0], [2, 0, 0], [1, 0], [0], [1, 0], [1, 1], [2, 1, 0], [1, 1], [0], [1, 0], [2, 0, 0], [2, 1, 0], [2, 1, 1], [1, 1], [2, 1, 0], [2, 0, 0], [2, 0, 1], [1, 0], [2, 0, 0], [2, 0, 1], [2, 1, 1], [2, 1, 0], [2, 1, 1], [2, 0, 1], [2, 0, 0].

Второй вариант (27 переходов, соответствует порядку *OKUC*): [0], [1, 0], [0], [1, 0], [1, 1], [1, 0], [2, 0, 0], [1, 0], [1, 1], [0], [1, 0], [2, 0, 0], [2, 1, 0], [2, 0, 0], [2, 0, 1], [2, 0, 0], [2, 1, 0], [1, 1], [2, 1, 0], [1, 1], [2, 1, 0], [2, 1, 1], [2, 1, 0], [2, 1, 1], [2, 0, 1], [1, 0], [2, 0, 0], [2, 0, 1], [2, 1, 1], [1, 1].

Следствие из рассмотренных примеров. Разница в эффективности, показанная в данных примерах невелика, что обусловлено простотой модели (малое количество состояний и дуг), а также оптимальностью последовательного использования сценариев.

Разница достигается исключительно за счет изменения расположения сценариев в обойме обходчика.

Это указывает на необходимость анализа работы обходчика и применения результатов этого анализа при разработке модели, а также тестовых сценариев.

4. Краткое описание обходчиков UniTESK

В настоящий момент CTesK [6], один из инструментов, поддерживающих разработку тестов по технологии UniTESK, включает два обходчика, реализующих различные алгоритмы построения обхода графов и имеющих различные ограничения на вид графов, с которыми они могут работать.

Первый обходчик, dfsm [7,8], реализует алгоритм генерации тестовой последовательности по неизбыточному описанию графа сценария при помощи построения его обхода в глубину и предназначен только для работы с детерминированными графами. Обход в глубину означает, что обходчик хранит цепочку из тех вершин, не все дуги которых пройдены. Попав в одну из таких вершин, он каждый раз доходит до конца цепочки, прежде чем пройти по еще не пройденной дуге. Тем самым, обходчик dfsm не следует рекомендации, сформулированной выше. Для вершин, где все дуги пройдены, этот обходчик хранит одну дугу, позволяющую (быть может, после прохождения еще каких-то дуг) попасть на цепочку вершин, имеющих еще не пройденные дуги.

Этот алгоритм прекрасно проявил себя в многочисленных проектах [9] по тестированию различных видов программного обеспечения при помощи технологии UniTESK. Однако требование детерминированности графа сценария требует от разработчиков тестов дополнительных усилий, направленных на избавление от недетерминизма графа, динамически извлекаемого из наблюдений за поведением тестируемой системы. Причем такой недетерминизм появляется достаточно часто, и не только из-за недетерминированности самой тестируемой системы, но и из-за

неаккуратного абстрагирования в графе сценария от деталей реализации. В то же время, в большинстве случаев только небольшая часть дуг графа соответствует недетерминированным переходам, и существует детерминированный остовный подграф, который мог бы быть использован для построения обхода графа.

Задача построения алгоритмов строящих обход недетерминированных графов, заданных неизбыточным образом, была успешно решена [10]. Но предложенный в работе [10] алгоритм является полновесным средством работы с существенно недетерминированными графами. В то же время, в большинстве случаев можно обойтись более простым решением, обладающим более узкой областью применимости, которое реализовано во втором обходчике CTesK, ndfsm [11].

ndfsm реализует жадный алгоритм построения обхода дуг графа, опираясь при этом на предположение о наличии детерминированного полного остовного подграфа, т.е. детерминированного подграфа, в который входят все вершины исходного графа. Жадный алгоритм, оказавшись в вершине, где все дуги уже пройдены, пытается найти ближайшую к ней вершину, где еще есть непройденная дуга, и попасть в нее. Значит, ndfsm действует в соответствии с высказанной рекомендацией. Наличие детерминированного остова позволяет обходчику всегда находить путь, ведущий в нужную вершину.

5. Сравнение эффективности обходчиков UniTESK

В данном разделе приводятся экспериментальные данные о сравнительной эффективности двух обходчиков CTesK.

В первой его части исследовалась зависимость производительности обходчиков от порядка обращений к сценарным функциям для небольших графов.

В следующих частях сравнивается эффективность работы обходчиков на разнообразных графах. Сравнивалось, в основном количество проходов по дугам графа (или же обращений к сценарным функциям), выполняемых обходчиками при построении обхода. При этом выполнение каждой отдельной сценарной функции максимально облегчено, в его ходе производился минимум действий. Иногда для наглядности приведено общее время работы теста, которое измерялось в секундах с допустимой погрешностью в 1 секунду.

При сравнении использовался компьютер с процессором AMD 3200+, 1024 MB памяти.

5.1. Зависимость работы обходчика от порядка сценарных функций

В Таблице 1 показано количество переходов, выполняемое обходчиками для построения обхода графов из раздела 3 при различных упорядочениях сценарных функций.

C в этой таблице обозначает операцию создания нового потока, *K* — уничтожение одного из имеющихся потоков, *U* — помещение функции в стек обработчиков завершения потока, *O* — выталкивание функции из этого стека.

Порядок сценарных функций	Простой пример (Рис. 2) Количество состояний — 4. Количество дуг — 8.		Сложный пример (Рис. 3) Количество состояний — 7. Количество дуг — 13.	
	ndfsm	**dfsm**	**ndfsm**	**dfsm**
CKUO	16[1]	22	31	76
CKOU	16	22	29	76
CUKO	16	25	29	76
CUOK	13[2]	25	27	88
COKU	16	19	30	88
COUK	13	25	27	88
KCUO	16	22	31	68
KCOU	16	22	29	68
KUCO	16	22	30	60
KUOC	16	22	30	60
KOCU	16	22	31	68
KOUC	16	22	31	59
UCKO	16	25	29	64
UCOK	13	25	27	69
UKCO	16	25	29	60
UKOC	16	25	29	60
UOCK	13	25	27	69
UOKC	16	25	31	69
OCKU	16	19	30	88
OCUK	13	25	27	88
OKCU	16	19	30	61
OKUC	16	19	31	55
OUCK	13	25	27	68
OUKC	16	25	31	68

Таблица 1. Длина строящегося обхода в зависимости от порядка сценарных функций.

Таблица 1 показывает, что, меняя только порядок обращений к сценарным функциям, можно добиться следующего ускорения более 35% для dfsm (сложный пример, сравнение *CUOK* и *OKUC*) и более 10% для ndfsm (сложный пример, *CKUO* и *CUOK*).

[1] Соответствует второму варианту из второго примера в разделе 3.
[2] Соответствует первому варианту из второго примера в разделе 3.

На рассмотренных примерах ndfsm демонстрирует более высокую производительность. Далее мы рассмотрим дополнительные примеры, подтверждающие это.

5.2. Обход дерева

Сравним эффективность обходчиков при обходе полного бинарного дерева высотой *N*, с операциями «опуститься на уровень ниже влево», «опуститься на уровень ниже вправо» и «подняться на уровень выше».

Теоретическая оценка длины обхода снизу равна количеству дуг $2^{N+2}-4$. Эта оценка может быть достигнута, например, при обходе в симметричном порядке [12].

В Таблице 2 приведены длины обходов и время их выполнения для разных обходчиков.

N	**dfsm, длина**	**dfsm, время**	**ndfsm, длина**	**ndfsm, время**	**мин. длина**
9	4088	6	2044	3	2044
10	8184	25	4092	12	4092
11	16376	100	8188	45	8188
12	32760	427	16380	190	16380

Таблица 2. Длина и время обхода дерева для разных *N*.

Обход дерева любопытен тем, что на нем при использовании обходчика ndfsm достигается минимальная длина обхода. Обходчик dfsm делает ровно в два раза больше проходов по дугам.

5.3. Обход полного графа и его модификаций

В данном разделе будут рассмотрены экспериментальные данные по построению обхода для следующих видов графов.

- Полный ориентированный граф с *N* вершинами, обозначаемый K_N.
- Граф $K_M(K_N)$, представляющий собой соединенные с помощью K_M *M* полных графов K_N. Вершины этих *M* графов пронумерованы числами от 0 до (*N*-1) и все вершины с номером 0 соединены между собой полным графом.
- Граф $K_M \times K_N$, являющийся декартовым произведением полных графов. Он тоже представляется как *M* графов K_N с пронумерованными вершинами, в которых в сем вершинам с одинаковыми номерами соединены друг с другом.

В следующих таблицах приведены длины и время (в секундах) построения обхода разными обходчиками для таких графов.

Числа *N* и *M* выбраны нечетными, потому что для полного графа с нечетным количеством вершин существует эйлеров цикл.

Теоретическая оценка длины обхода K_N снизу равна количеству его дуг $E(K_N) = N \cdot (N-1)$.

Граф	dfsm, длина	dfsm, время	ndfsm, длина	ndfsm, время	мин. длина
K_3	14	0	8	0	6
K_5	60	0	24	0	20
K_7	154	0	48	0	42
K_{51}	46750	9	2600	1	2550
K_{53}	52364	11	2808	1	2756
K_{55}	58410	12	3024	1	2970
K_{57}	64904	13	3248	1	3192
K_{59}	71862	15	3480	1	3422
K_{61}	79300	17	3720	1	3660

Таблица 3. Длина и время обходов полных графов.

Отметим тот факт, что обходчик ndfsm проходит на (N-1) дуг больше, чем нужно по минимуму для обхода полного графа K_N.

Оценка длины обхода графа $K_M(K_N)$ снизу тоже равна количеству его дуг $E(K_M(K_N)) = M \cdot E(K_N) + E(K_M) = M \cdot N \cdot (N-1) + M \cdot (M-1)$.

Граф	dfsm, длина	dfsm, время	ndfsm, длина	ndfsm, время	мин. длина
$K_3(K_{51})$	140264	29	7810	2	7656
$K_5(K_{51})$	233810	53	13028	3	12770
$K_7(K_{51})$	327404	80	18254	5	17892
$K_3(K_{53})$	157106	35	8434	2	8264
$K_5(K_{53})$	261880	60	14068	4	13800
$K_7(K_{53})$	366702	94	19710	6	19334
$K_3(K_{55})$	175244	40	9082	2	8916
$K_5(K_{55})$	292110	71	15148	4	14870
$K_7(K_{55})$	409024	108	21222	6	20832

Таблица 4. Длина и время обходов графов $K_M(K_N)$.

Отметим, что длина обхода этих графов с помощью dfsm может быть вычислена по той же формуле, что и количество дуг: если $D(G)$ обозначает длину обхода графа G с помощью dfsm, то $D(K_M(K_N)) = M \cdot D(K_N) + D(K_M)$.

Можно предположить, что при обходе графа, состоящего из нескольких слабо связанных (при помощи одной-двух дуг) частей, длина его обхода получается сложением длин обходов частей и количества проходов по связывающим дугам.

Оценка длины обхода графа $K_M \times K_N$ снизу также равна количеству его дуг $E(K_M \times K_N) = M \cdot E(K_N) + N \cdot E(K_M) = M \cdot N \cdot (N+M-2)$.

Граф	dfsm, длина	dfsm, время	ndfsm, длина	ndfsm, время	мин. длина
$K_3 \times K_{51}$	269265	62	8108	1	7956
$K_5 \times K_{51}$	804614	186	14024	5	13770
$K_7 \times K_{51}$	1694413	406	20348	5	19992
$K_3 \times K_{53}$	301095	68	8744	2	8576
$K_5 \times K_{53}$	898884	216	15104	4	14840
$K_7 \times K_{53}$	1890675	302	21888	5	21520
$K_3 \times K_{55}$	335337	50	9404	2	9240
$K_5 \times K_{55}$	1000234	164	16224	3	15950
$K_7 \times K_{55}$	2101501	365	23484	5	23100

Таблица 5. Длина и время обходов графов $K_M \times K_N$.

Теперь длина обхода с помощью dfsm уже не может быть определена с помощью аналогичной формулы.

Приведенные таблицы показывают, что на многих графах обходчик ndfsm работает эффективнее dfsm, и, если проверка детерминизма графа не является необходимым элементом тестирования, предпочтительнее использовать первый обходчик.

Можно заметить, что при увеличении количества дуг длина обхода с помощью dfsm составляет порядка 50% от произведения количества дуг на количество состояний, что означает практическую невозможность использования dfsm-обходчика в случае насыщенных графов с сотнями состояний.

6. Заключение

Моделирование поведения практически важной системы в виде конечного автомата часто приводит к большому числу состояний и переходов в итоговой модели. При тестировании такой системы с помощью технологии UniTESK выполняется автоматическое построение обхода графа состояний и переходов модели. Длина такого обхода в значительной степени зависит от используемого алгоритма обходчика.

В работе было проведен сравнительный анализ эффективности построения обхода графов с помощью обходчиков dfsm и ndfsm, входящих в инструмент CTesK. Полученные результаты показывают, что для многих графов второй обходчик строит существенно более короткие обходы, и, соответственно, позволяет получать тесты, работающее значительно быстрее, но обеспечивающие те же значения тестового покрытия.

Кроме того, показана возможность некоторого уменьшения времени работы теста только за счет изменения очередности задания сценарных функций.

Литература

[1] В. В. Кулямин, А. К. Петренко, А. С. Косачев, И. Б. Бурдонов. *Подход UniTesK к разработке тестов.* Программирование, т. 29, № 6, стр. 25–43, 2003.

[2] А. В. Баранцев, И. Б. Бурдонов, А. В. Демаков, С. В. Зеленов, А. С. Косачев, В. В. Кулямин, В. А. Омельченко, Н. В. Пакулин, А. К. Петренко, А. В. Хорошилов. *Подход UniTesK к разработке тестов: достижения и перспективы*. Труды Института системного программирования РАН, т. 5, стр. 121–156, 2005.

[3] http://www.opengroup.org/onlinepubs/007908799/xsh/threads.html

[4] http://www.opengroup.org/onlinepubs/009695399/functions/pthread_cleanup_pop.html

[5] http://www.unix.org/version3/

[6] http://www.unitesk.com/content/category/7/14/33/

[7] И. Б. Бурдонов, А. С. Косачев, В. В. Кулямин. *Использование конечных автоматов для тестирования программ.* Программирование, т. 26, № 2, стр. 61–73, 2000.

[8] И. Б. Бурдонов, А. С. Косачев, В. В. Кулямин. *Неизбыточные алгоритмы обхода ориентированных графов: детерминированный случай.* Программирование, т. 29, № 5, стр. 59–69, 2003.

[9] http://www.unitesk.com/content/category/8/20/54/

[10] И. Б. Бурдонов, А. С. Косачев, В. В. Кулямин. *Неизбыточные алгоритмы обхода ориентированных графов: недетерминированный случай.* Программирование, т. 30, № 1, стр. 2–17, 2004.

[11] А. В. Хорошилов. *Отчет о научно-исследовательской работе. Алгоритм обхода недетерминированных графов, обладающих детерминированным полным остовным подграфом.* Шифр РИ-19.0/002/216.

[12] http://www.structur.h1.ru/derevo.htm